UNIVERSITY OF OREGON

CIS 415: Operating Systems

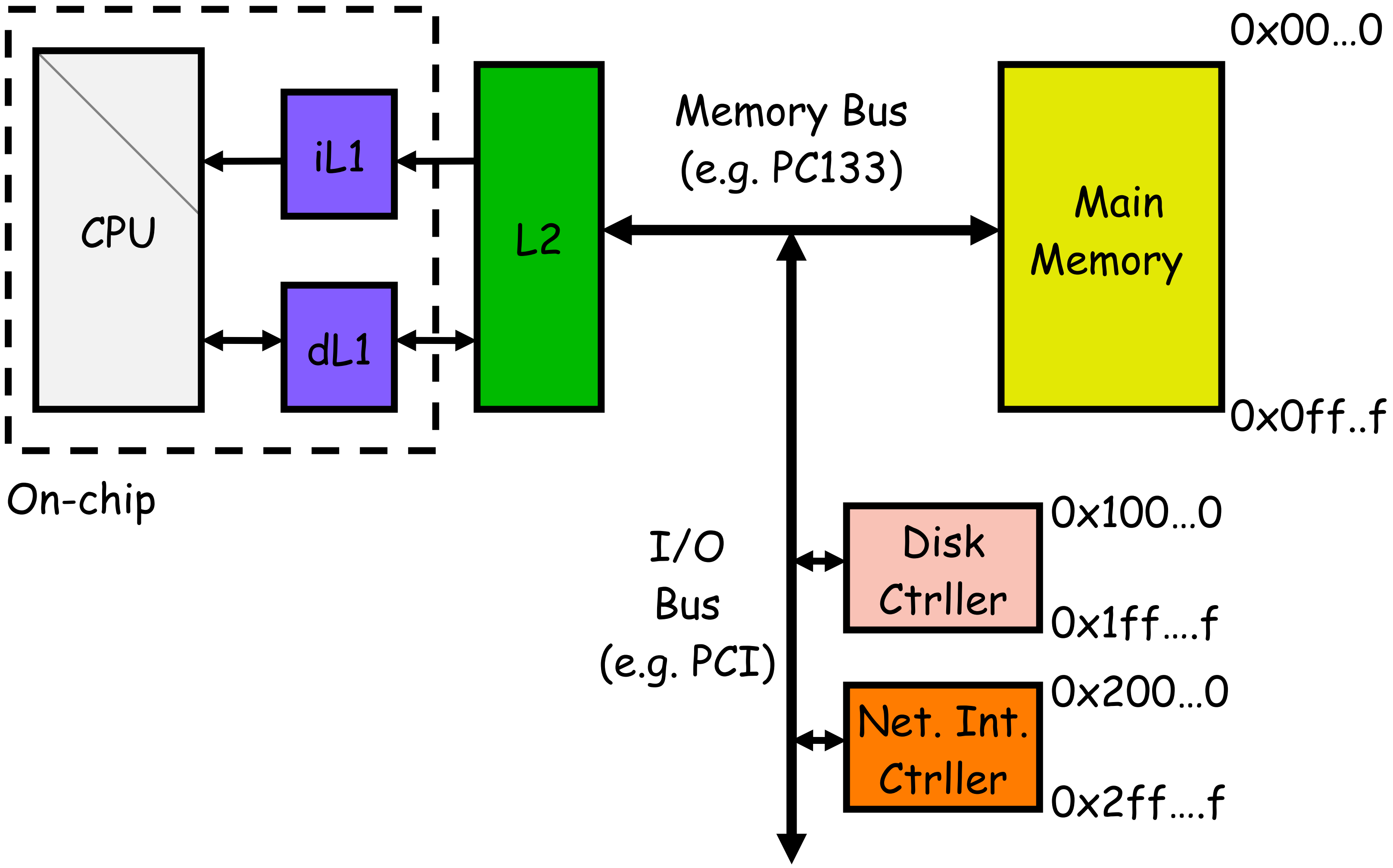
I/O Peripherals

Spring 2012
Prof. Kevin Butler

OS role in I/O

- Share the same device across different processes/users
- User does not see the details of how hardware works
- Device-independent interface to provide uniformity across devices.

I/O Peripherals



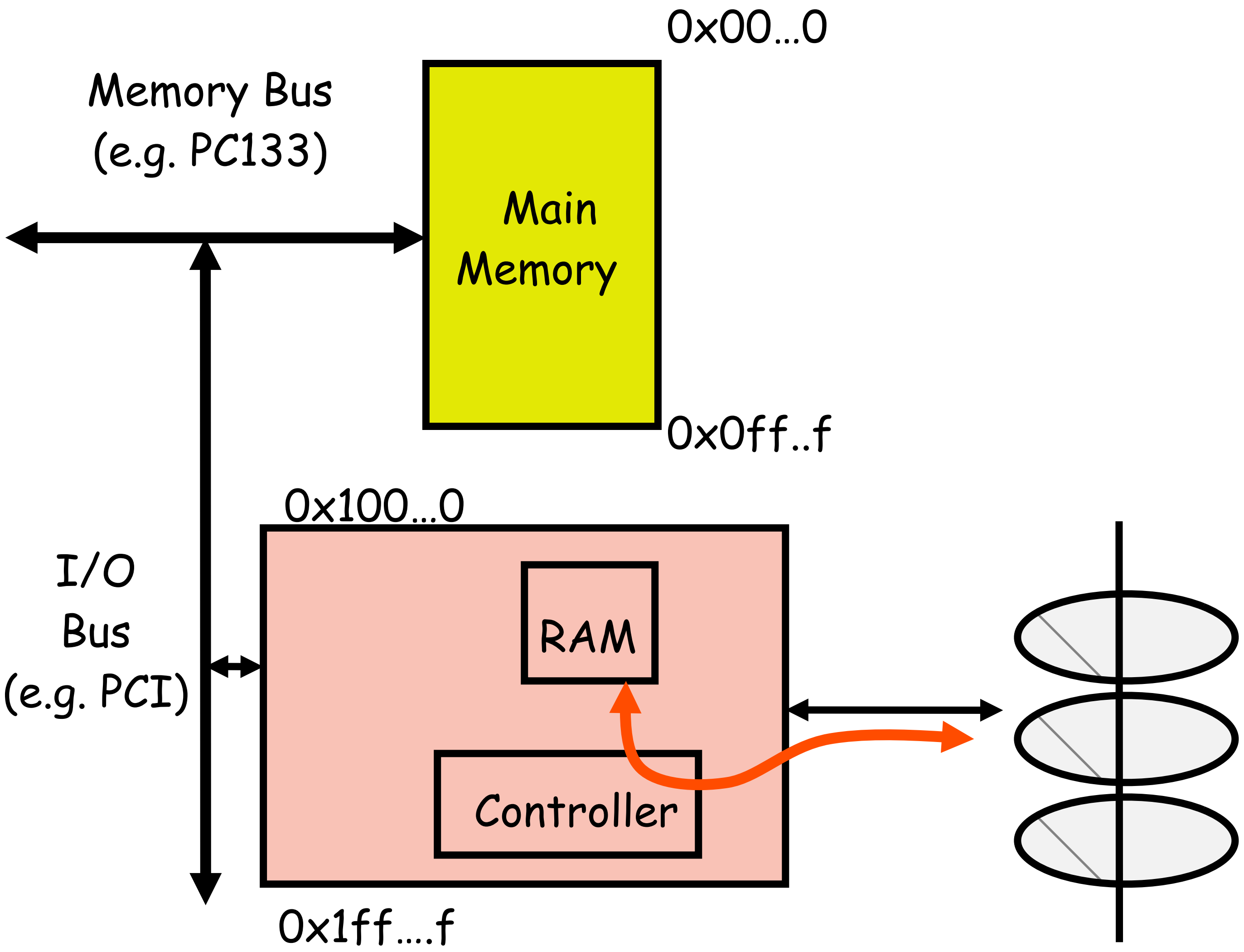
Talk to Devices

- Communication
  - Send instructions to the devices
  - Get the results
- I/O Ports
  - Dedicated I/O registers for communicating status and requests
- Memory-mapped I/O
  - Map the registers into address space
  - Communicate requests through memory operations
- Memory-mapped data "registers" can be larger
  - Think graphics device

Memory-mapped I/O

- Can read and write device registers just like normal memory.
- However, user programs are NOT typically allowed to do these reads/writes.
- The OS has to manage/control these devices.
- The addresses to these devices may not need to go through address translation since
  - OS is the one accessing them and protection does not need to be enforced, and
  - there is no swapping/paging for these addresses.

Consider a disk device …



Reading sector from disk

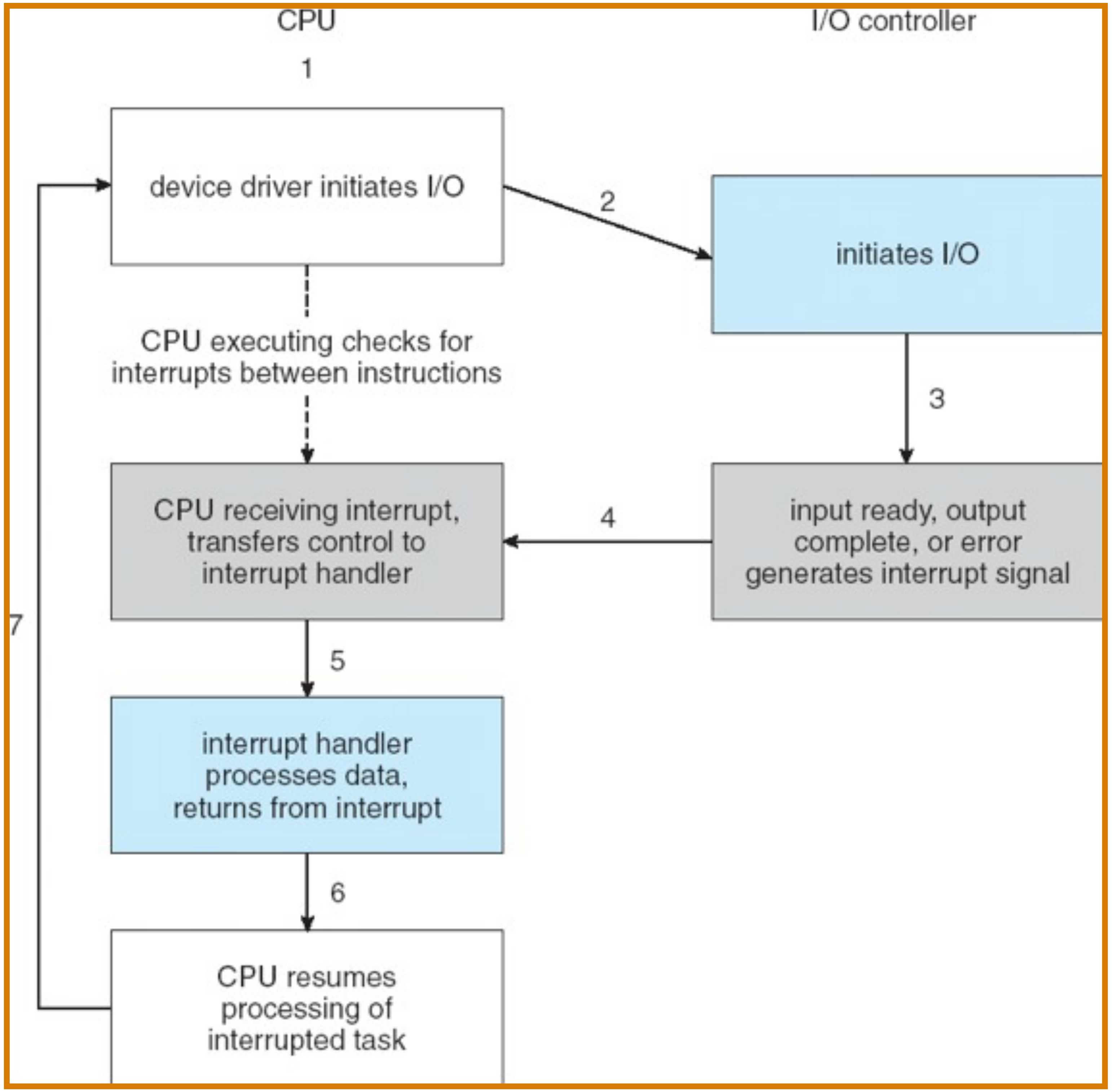
```
Store [Command_Reg], READ_COMMAND
Store [Track_Reg], Track #
Store [Sector_Reg], Sector #

/* Device starts operation */

L:   Load R, [Status_Reg]
     cmp R, 0
     jeq
```

You don’t want to do this! Instead, block/switch to other process and let an interrupt wake you up.

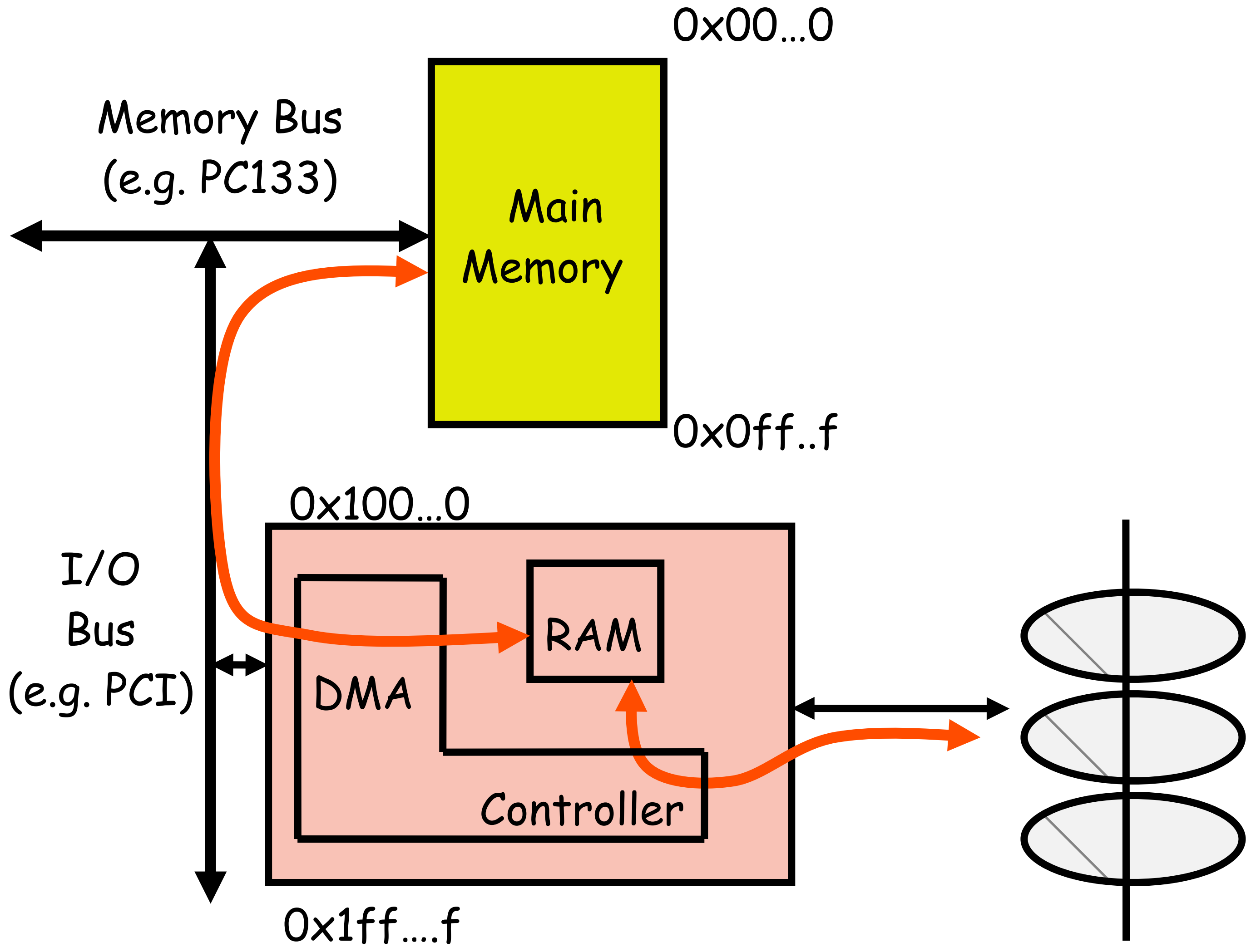
```
/* Data now on memory of card */

For i = 1 to sectorsize
    Memtarget[i] = MemOnCard[i]
```

This is again a lot of overhead to ask the main CPU to do!

Interrupt Cycle



DMA engine



Used to offload work of copying
Lots of different offload engines possible in systems

```
Store [Command_Reg], READ_COMMAND
Store [Track_Reg], Track #
Store [Sector_Reg], Sector #
Store [Memory_Address_Reg], Address

/* Device starts operation */

P(disk_request);

/* Operation complete and data is
now in required memory locations*/
```

Assuming an integrated DMA & disk controller.

Called when DMA raises interrupt after Completion of transfer

```
ISR() {
V(disk_request);
}
```

Issues to consider

- What is purpose of RAM on card?
  - To address the speed mismatch between the bit stream coming from disk and the transfer to main memory.
- When we program the DMA engine with address of transfer (Store [Memory_Address_Reg], Address), is Address virtual or physical?
  - It has to be a physical address, since the addresses generated by the DMA do NOT go through the MMU (address translation).
  - But since it is the OS programming the DMA, this is available and it is NOT a problem.
  - You do NOT want to give this option to user programs.
  - Also, the address needs to be "pinned" (cannot be evicted) in memory.

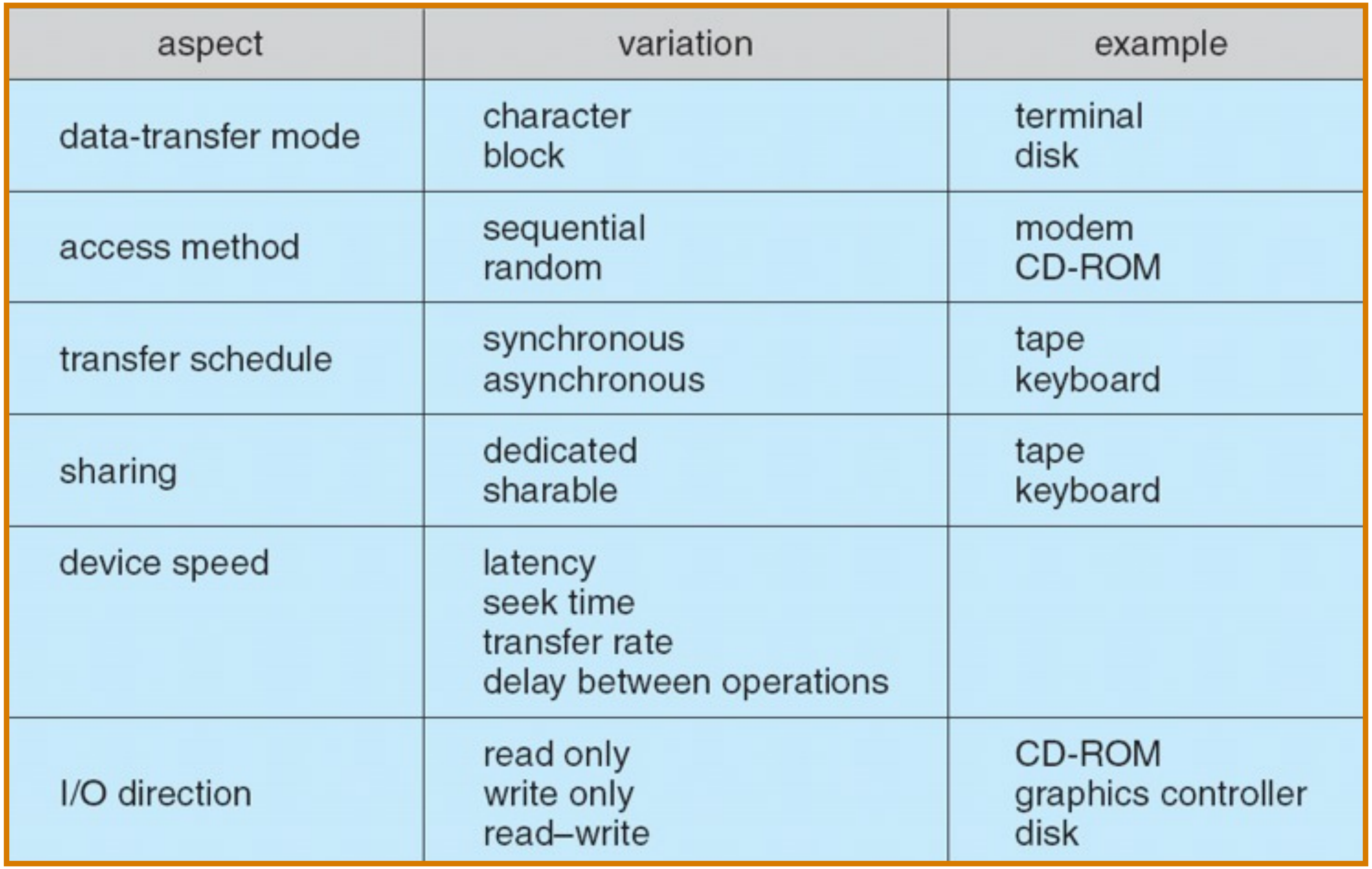
I/O Devices

- Block devices:
  - usually stores information in fixed size blocks
  - you read or write an individual block independently of others by giving it an address.
  - E.g., disks, tapes, …
- Character devices:
  - delivers or accepts streams of characters
  - Not addressable.
  - E.g., terminals, printers, mouse, network interface.

Principles of I/O Software

- Provide device independence:
    - same programs should work with different devices.
    - uniform naming -- i.e., name shouldn't depend on the device.
    - error handling, handle it as low as possible and only if unavoidable pass it on higher.
    - synchronous (blocking) vs. asynchronous (interrupt driven). Even though I/O devices are usually async, sync is easier to program

Device Characteristics



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

I/O Software

- A layered approach:
  - Lowest layer (device dependent): Device drivers
  - Middle layer: Device independent OS software
  - High level: User-level software/libraries
- The first 2 are part of the kernel.

Device Drivers

- Accept abstract requests from device-independent OS software, and service those requests.
- There is a device driver for each "device"
- However, the interface to all device drivers is the same.
  - Open(), close(), read(), write(), interrupt(), ioctl(), …

Disk driver

```
Semaphore request;

Open() { ….}

Close() { … }

Read() {
    ….
    program the device
    P(request);
    …
}

Write() { ….}
```

```
Interrupt() {
    check what caused the interrupt
    case disk_read:V(request);
    …
}
```

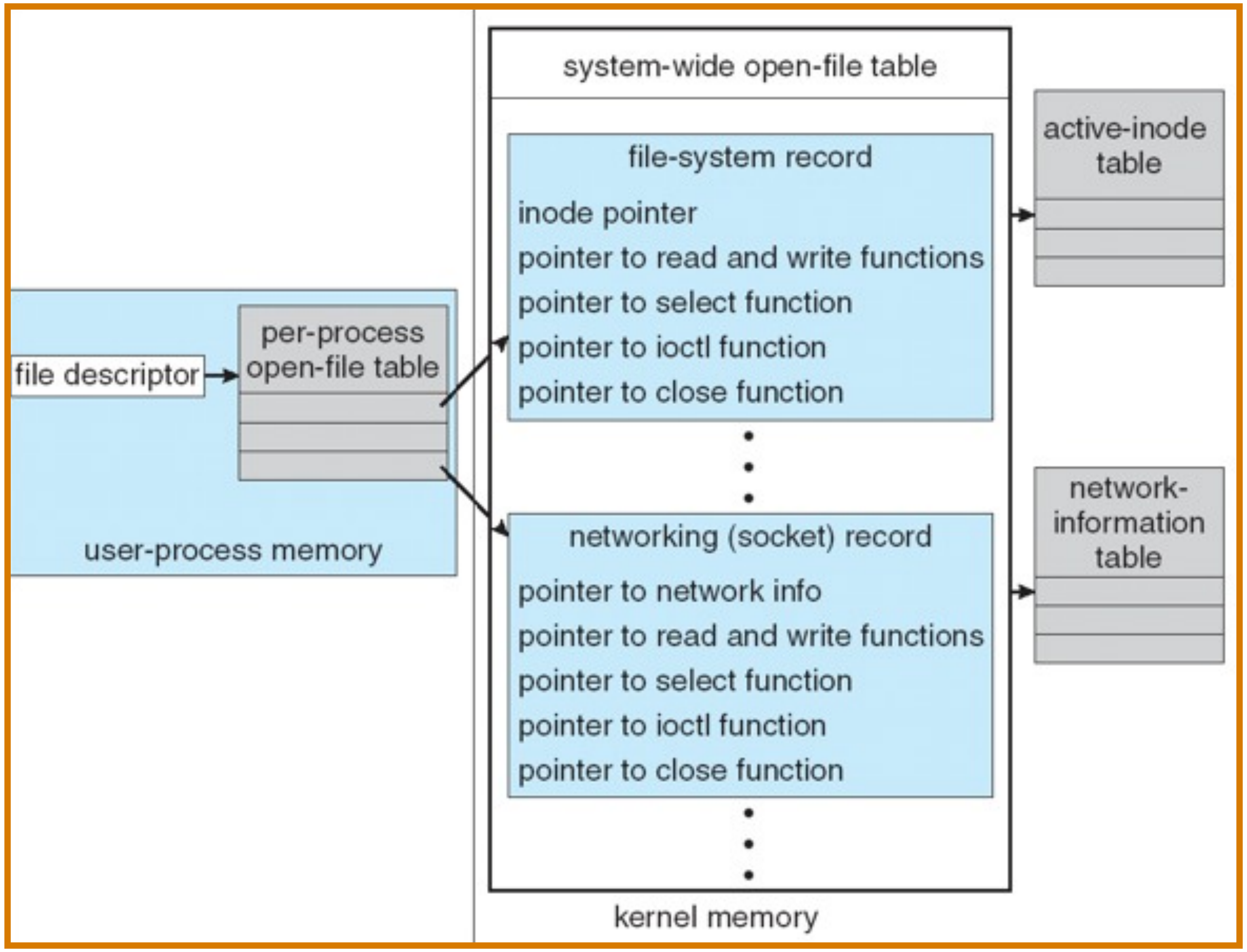
Device-independent OS Layer

- Device naming and protection
  - Each device is given a (major #,minor #) – present in the i-node for that device
  - Major # identifies the driver
  - Minor # is passed on to the driver (to handle sub-devices)
- Does buffering/caching
- Uses a device-independent block size
- Handles error reporting in a device-independent fashion.

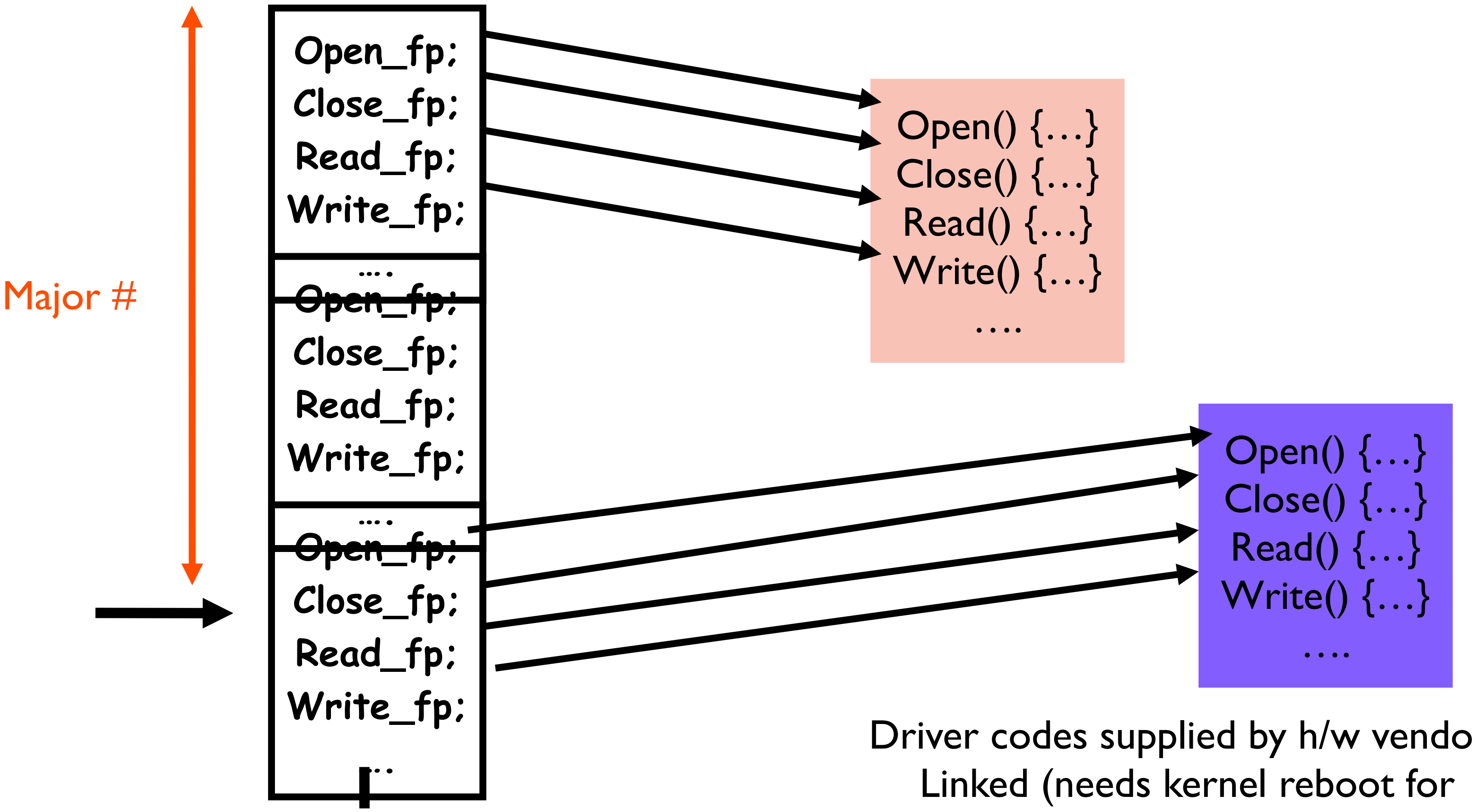
Putting things together (UNIX)

- User calls open("/dev/tty0","w") which is a system call.
- OS traverses file system to find the i-node of tty0.
- This should contain the (major #, minor #).
- Check permissions to make sure it is allowed.
- An entry is created in OFDT, and a handle is returned to the user.
- When user calls write(fd, ….) subsequently, index into OFDT, get major/minor #s.

I/O and Kernel Objects



Getting to the driver routine



In Memory Data Struct

Driver codes supplied by h/w vendors
Linked (needs kernel reboot for
Installation) or dynamically loaded into
Kernel.

Driver I/O

- Copy the bytes pointed to by the pointer given by user, into a kernel "pinned" (which is not going to be paged out) buffer.
- Use the above data structure, to find the relevant driver's write() routine, and call it with the pinned buffer address, and other relevant parameters.
- For a write, one can possibly return back to user even if the write has not propagated. On a read (for an input device), the driver would program the device, and block the activity till the interrupt.

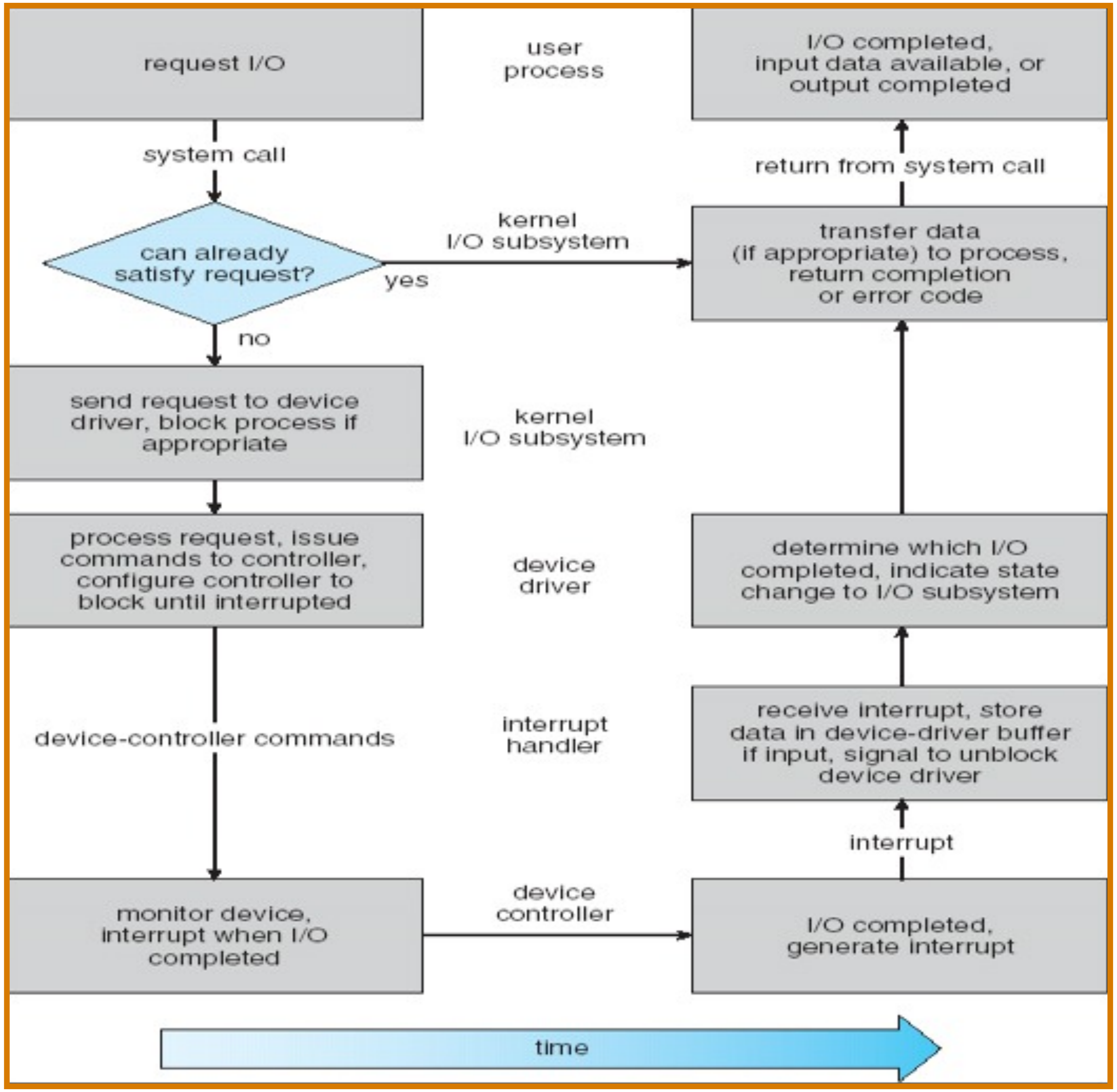
Block Devices

- Previous description was for *character device*
- In a *block device*, before calling the driver, check the buffer cache that the OS is maintaining to see if the request can be satisfied before going to the driver itself.
- The lookup is done based on *(major #, logical block id)*.
- Thus it is a unified device-independent cache across all devices.

Generality of I/O

- This is all for the user referring to an I/O device (`/dev/*`).
- Note: It is not very different when the user references a normal file. In that case, we have already seen how the file system generates a request in the form of a logical block id, which is then sent to the driver where the specified file system resides (disk/CD/…)

Life Cycle of an I/O Request



request I/O

user process

I/O completed, input data available, or output completed

system call

return from system call

can already satisfy request?

yes

kernel I/O subsystem

transfer data (if appropriate) to process, return completion or error code

no

send request to device driver, block process if appropriate

kernel I/O subsystem

process request, issue commands to controller, configure controller to block until interrupted

device driver

determine which I/O completed, indicate state change to I/O subsystem

device-controller commands

interrupt handler

receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

interrupt

monitor device, interrupt when I/O completed

device controller

I/O completed, generate interrupt

time

Summary

- Input/Output
    - The OS Manages Device Usage
    - Communication
    - I/O Subsystem

Next day

- Wrap-up and presentations